ANSI C Cryptographic API Profile
for AES Candidate Algorithm Submissions

Original: January 16, 1998
Revision 1: February 6, 1998
Revision 2: March 19,1998
Revision 3: April 6,1998

1. Overview

This document specifies the ANSI C interface profile for implementations of AES candidate algorithms. C implementations shall support the syntax and parameterization of the interface profile messages as described in this API.

(4/6/98) Some changes were made to the ANSI C language profile to simplify and clarify its usage. A few are minor bug fixes and others are changes to the function calls and parameters in the key generator interface. Also, the ANSI C profile was extracted from the existing document and is now presented in this separate document.

2. Key Generation Interface

Each AES submitter will be required to implement this interface because NIST anticipates that some candidate algorithms will have unique requirements for and methods of key generation. Implementations shall support generation of 128, 192, and 256-bit keys, and, where applicable, are responsible for controlling the key generation process to avoid the possibility of creating the equivalent of weak keys for a given algorithm.

The ANSI C key generation programming interface uses two structures and a set of functions to generate and manipulate key. The first structure, *keyInstance*, is an algorithm independent structure that shall remain unchanged for all submissions. The second structure, *keyParams*, is algorithm dependent and is used to contain any algorithm-specific key material or information that is necessary. **All implementations must be sure to document any algorithm-specific parameters and their use.**

The key generator functions get used in this way. First, *keyInit()* is called with the appropriate parameters which get loaded into the *keyInstance* structure and the *keyParams* structure. If the candidate algorithm has any key independent setup to perform, it should be performed in this routine. Second, *keySetup()* is called to perform any key specific setup that is necessary. The *generateRandomKey()* function is provided to allow implementations to generate arbitrary keys. This function could also be used as a pseudo-random number generator. You are NOT required to include a pseudo-random number generating (PRNG) function in your submission; however, the use of the algorithm as a PRNG can be a bonus as far as flexibility is concerned (see Section 2.B.6). For the purpose of the KAT and MCT tests, random keys will not be generated. If your algorithm can be used to generate pseudo-random values for keys, you may wish to include this feature in your submission. Since some of our testing at NIST will involve looking at the pseudo-random number generation capabilities of algorithms, this feature will be useful to us. Once again, though, it is not required.

```
➢ typedef struct {
      BYTE        mode;
      int         keyLen;
      BYTE        *keyMaterial;  /* BYTE is defined as an unsigned char */
      BYTE        *random;
      keyParams   *params;
      } keyInstance;
```

(4/6/98 – mode type was changed from int to BYTE to match the cipherInstance structure. Also, additional modes were added below. randomSeed was renamed to random and moved from the keyparams structure to the keyInstance structure)

```
➢ typedef struct  {
      BYTE        *KS;           /* SAMPLE: key schedule, a la DES  */
      } keyParams;
```

❖ **keyInit**

int keyInit(keyInstance *key, int mode, int keyLen, BYTE *keyMaterial, BYTE *random, BYTE *params)

Initializes a keyInstance with the following information:

- keyLen: The key length (128, 192, 256, and possibly more) of the key,
- mode: one of the following: ENCRYPT_ECB=1, DECRYPT_ECB=2, ENCRYPT_CBC=3, DECRYPT_CBC=4, ENCRYPT_CFB1=5, DECRYPT_CFB1=6,
- The raw key data, keyMaterial, if present,
- An optional source of randomness (a seed), and
- Other params, if necessary.

Parameters:

<u>key</u>: a structure that holds the keyInstance information
<u>mode</u>: an integer value that indicates if the key is being set for encryption or decryption and in what mode (e.g., mode=Encrypt in cipher block chaining mode=3)
<u>keyLen</u>: an integer value that indicates the length of the key in bits.
<u>keyMaterial</u>: the raw key information (blocksize/4 ASCII characters representing the hex values for the key). This value may not be present. For example,

"0123456789abcdef0123456789abcdef" is the string for a key with the binary value: 000100100011010001010110011110001001101010111100 11011110...
random: an optional seed value for the generation of a random key. (Same sort of format as keyMaterial.)
params: an optional string for passing algorithm specific information. The format of the "params" string IS IMPLEMENTATION SPECIFIC, some algorithms will require different parameters. SPECIFY THE USE OF THIS STRING FOR YOUR IMPLEMENTATION.

Returns:

TRUE - on success
BAD_KEY_PARAMS - params structure not valid for this cipher
BAD_KEY_MAT - keyMaterial is invalid (e.g., wrong length)

- **keySetup**

int keySetup(keyInstance *key)

keyInstance should already have the keyMaterial loaded. Use this routine to perform any algorithm specific setup that is necessary, e.g., initialization of key schedule in DES.

Returns:

TRUE - on success
BAD_KEY_SPEC - didn't include keyMaterial to generate a key

- **generateRandomKey**

(4/6/98 – As stated above, this routine is not strictly needed for the KAT and MCT tests, but is included to allow for greater flexibility and to allow implementors to provide a means for generating pseudo-random data.)

int generateRandomKey(keyInstance *key)

Generates a random key. The value generated by this function is the raw key, e.g., "0123456789abcdef0123456789abcdef". The value for this key is stored in the key->keyMaterial field of the structure. This value shall be stored as the ASCII string representation of the key.

This will most likely use the contents of key->random as a seed to an algorithm for generating a pseudo-random secret key. Any algorithm specific setup will be performed in the keySetup() routine.

Returns:

TRUE - on success
FALSE - on failure

(4/6/98 – Deleted function called keyGetKey. Programs can get this info from key->keyMaterial directly.)

3. Cipher Object Interface

The ANSI C cipher programming interface uses two structures and a set of functions to manipulate cipher data. The first structure, *cipherInstance*, is an algorithm independent structure that shall remain unchanged for all submissions. The second structure, *cipherParams*, is algorithm dependent and is used to contain any algorithm-specific cipher data or information that is necessary. **All implementations must be sure to document any algorithm-specific parameters and their use.**

The cipher routines get used in this way. First, *cipherInit()* is called with the appropriate parameters in *params* to be loaded into the *cipherInstance* structure. *cipherInit()* will perform any additional algorithm setup that is required, e.g., incorporating the use of an Initialization Vector. Then blocks of data are supplied to either cipherUpdate() or *cipherFinal()* for ciphering. *cipherUpdate()* can be called multiple times and the *cipherInstance* structure is used to maintain state information between calls. *cipherFinal()* would then be called to cipher the final block or peace of a block. If the entire block is available for ciphering at one time, as is the case with the KAT and MCT tests, *cipherUpdate()* can be avoided and a single call to *cipherFinal()* can be used.

- ```
  typedef struct {
      BYTE  mode;              /*  ENCRYPT_ECB, DECRYPT_ECB, etc. */
      int   numBytes;          /*  Number of bytes processed by cipher */
      int   bufLen;            /*  Number of bytes in the buffer */
      BYTE  buffer[BLOCKSIZE]; /*  Unprocessed data */
      cipherParams  *params;   /*  Algorithm specific information */
      } cipherInstance;
  ```

- ```
  typedef struct {
      BYTE *IV;      /* Sample: Possible Initialization Vector for cipher */
      int   blocksize;   /*  Sample: If alg. handles additional block sizes */
      int   padMode;     /*  Sample: Padding mode of partial blocks */
  ```

} cipherParams;

- **cipherInit**

int cipherInit(cipherInstance *cipher, keyInstance *key, BYTE mode, cipherParams *params)

Initializes the cipher with the mode and sets the buffer information to empty. If any algorithm specific setup is necessary take care of that as well. Params may contain initialization information like an Initialization Vector. If the algorithm can use other block sizes than 128-bits, this value can be loaded in params as well. *(4/6/98 – removed the random parameter. If this is necessary or useful, include it in the cipherParams structure.)*

Parameters:

cipher – the cipherInstance being loaded
key – the ciphering key
mode - the operation mode of this cipher (this is one of ENCRYPT_ECB, DECRYPT_ECB, etc.)
params - the algorithm parameters (implementation defined)

Returns:

TRUE - on success
BAD_CIPHER_KEY - the key passed does not agree with this cipher (e.g., bad mode)
BAD_CIPHER_PARAMS - the params struct is invalid for this cipher

- **cipherUpdate**

int cipherUpdate(cipherInstance *cipher, keyInstance *key, BYTE *input, int inputOffset, int inputLen, BYTE *outBuffer)

Uses the cipherInstance object and the keyInstance object to encrypt or decrypt the data in the input buffer. This data need not be an integral block unit (e.g., 128 bits). The output (either the encrypted or decrypted data) is returned in outBuffer. The routine returns the number of bytes ciphered.

Parameters:

cipher – the cipherInstance to be used
key – the ciphering key
input - the input buffer

<u>inputOffset</u> - the offset in input where the input starts
<u>inputLen</u> - the input length
<u>outBuffer</u> – contains the ciphered data

Returns:
The number of bytes ciphered, or
BAD_CIPHER_STATE - cipher in bad state (e.g., not initialized)

- **cipherDoFinal**

int cipherDoFinal(cipherInstance *cipher, keyInstance *key, BYTE *input, int inputOffset, int inputLen, BYTE *outBuffer)

Uses the cipherInstance object and the keyInstance object to encrypt or decrypt the data in the input buffer. This data need not be an integral block unit (128 bits). Since this is the final call for an encryption or decryption operation, any non-integral block unit is padded to a full block and the ciphered. The output (either the encrypted or decrypted data) is return in outBuffer. The routine returns the number of bytes ciphered.

Parameters:
<u>cipher</u> – the cipherInstance to be used
<u>key</u> – the ciphering key
<u>input</u> - the input buffer
<u>inputOffset</u> - the offset in input where the input starts
<u>inputLen</u> - the input length
<u>outBuffer</u> – contains the ciphered data

Returns:
The number of bytes ciphered, or
BAD_CIPHER_STATE - cipher in bad state (e.g., not initialized)
BAD_CIPHER_BLOCK - cipher is block, but didn't receive full block and no padding requested